A Comparison Between Alice and Elizabeth Chatbot Systems

Technical Report · February 2004

Source: CiteSeer

2 authors:

Bayan Abu Shawar
Al Ain University

64 PUBLICATIONS   1,737 CITATIONS

Eric Atwell
University of Leeds

435 PUBLICATIONS   5,427 CITATIONS

University of Leeds

SCHOOL OF COMPUTING

RESEARCH REPORT SERIES

Report 2002.19

A Comparison Between
Alice and Elizabeth Chatbot Systems

by

Bayan Abu Shawar & Eric Atwell

December 2002

Abstract

This study examines two chatter bots systems called ALICE and Elizabeth, which are adapted from ELIZA program. Joseph Weizenbaum implemented ELIZA in 1966 and it was originally designed to emulate a psychotherapist.

This report also provides an introduction to the analysis of ALICE and Elizabeth focusing in the knowledge representation and pattern matching algorithms for each one of them. The report then illustrates the main differences between them and concludes that it will be easier to build machine learning for ALICE because of its simple pattern matching techniques than building one for Elizabeth since it depends on rules.

Introduction

A chatbot is a computer program aimed at simulating the conversation of a human being.

Section one of this research presents a study about ALICE chatterbox system, which is an abbreviation for Artificial Linguistic Internet Computer Entity that is implemented by Dr. Richard S. Wallace in 1995. Section Two displays a study for Elizabeth chatterbox system that is implemented by Dr. Peter Millican in University of Leeds. Both sections discuss how knowledge is represented within each system, displaying some necessary information about input, output patterns, how they are used and how the interpreter of each system do pattern matching to generate answers. Finally, section three illustrates the comparison between these systems.

1. ALICE System

ALICE: the Artificial Linguistic Internet Computer Entity. Is a robot that you can make chatting with it. ALICE knowledge is stored in AIML files.

AIML is an abbreviation of The Artificial Intelligent Mark up Language that is a derivative of Extensible Mark up Language(XML). The next sections describe necessary information about AIML elements, its categories, how they are used in ALICE and Pattern Matching Algorithm.

1.1 AIML Files

Each AIML file start with an <aiml> tag represent the AIML version being used, and it contains the AIML elements which consists of data objects called AIML objects. These objects are made up of units called **topics** and **categories**, which contain either parsed or unparsed data.

The topic is an optional top level element, has a name attribute and a set of categories related to that topic. Each category contains a pattern represent the user input and a template implies robot response.

The full AIML format with topic is:

< aiml version=”1.0” >
< topic name=” the topic” >

<category>
<pattern>PATTERN</pattern>
<that>THAT</that>
<template>Template</template>
</category>
    ..
    ..
    ..
</topic>
</aiml>

Note that:
- Each element has an open and close tag.
- <that> tag is optional depends if current user input(pattern) depends on a previous bot output.

1.2 Types of Categories

There are three types of categories:
1- Atomic Category.   2- Default Category   3- Recursive Category.

1- **Atomic Categories:** are those with patterns that does not have wildcards “*” or “_”.

    <category>
    <pattern>10 DOLLARS**</pattern>**
    <template> wow, what a cheap**</template>**
    </category>

    In the above category:

    If the user Input: 10 dollars
    Then root output: wow, what a cheap.

2- **Default Categories:** are those with patterns has a wildcards “*” or “_”. These patterns results from a reduction process while the robot searching for the appropriate matching. These wildcards match any input but they are differ in their alphabetical order.
Assume the previous category, if the robot does not find the previous atomic pattern, then he will try to find the following default pattern:

```
<category>
<pattern>10 *</pattern>
<template> It is ten.</template>
</category>
```

So robot output will be: It is ten.

3- **Recursive categories**: It is a property of template not pattern. The template calls the pattern matcher recursively using <srai> and <sr> tags which refers to Simply recursive artificial intelligence and symbolic reduction.

Recursive categories: have many application:
   1- Mapping inputs to another inputs in order to:
      a. Simplify the language.
      b. Map multiple patterns to the same response so it solve the problem of synonymous patterns.
   2- Partitioning: break down an input into one or more parts, and then combine their responses back together.

1.a Simplification user input to a simpler pattern form

For example, in English there are different ways to ask about X:
   • Describe x?
   • Tell me about X?
   • Do you know what X is?
The knowledge stored in the simplest way, so whatever the question is, it will be reduced to category like <What is >.

Assume the user input : WHAT DO YOU KNOW ABOUT NEURAL NETWORKS
Bot output: A neural network is a model of computation based on neuron cells.

Actually the following categories are used in matching:

```
<category>
<pattern>WHAT DO YOU KNOW ABOUT NEURAL NETWORKS</pattern>
<template>
<srai>WHAT IS A NEURAL NETWORK</srai>
</template>
</category>
```

In the previous category he found the atomic pattern then simplifying it to a new form, and search for this new input again, and match it with:

```
<category>
<pattern>WHAT IS A NEURAL NETWORK</pattern>
<template>
A neural network is a model of computation based on neuron cells.
</template>
</category>
```

Consider the following example that deals with synonymous patterns and partitioning.

Assume you have the following categories:

(1)   <category>
        <pattern>_ HOW ARE YOU</pattern>
        <template>
        <sr/> <srai>HOW ARE YOU</srai>
        </template>
        </category>

(2)   <category>
        <pattern>HALO</pattern>
        <template>
        <srai>HELLO</srai>
        </template>
        </category>

(3)   <category>
        <pattern>HELLO</pattern>
        <template>
        <random>
        Well hello there!
        Hi there!
        Hi there. I was just wanting to talk
        Hello there !
        </random>
        </template>
        </category>

(4)   <category>
        <pattern>HOW ARE YOU</pattern>
        <template>
        <random>
        Everything is going extremely well.
        I'm doing fine thanks how are you?
        Everything is running smoothly.
        </random>
        </template>
        </category>

If the user Input: halo how are you?
Then Root output: hi there! I'm doing fine thanks how are you?

The process is done as follows:

1- This input will match category(1) that partition it into two sentences:
   Sentence one: represent by <sr/> tag that match (_) which is the word HALO.
   Sentence two: HOW ARE YOU that is the result of reduction operation to the original
   input by using <srai> tag then he will scan for these two patterns.

2- An atomic patterns will find for HALO which is substituted by HELLO in category(2) and match it again with category (3) and the answer will be selected randomly from the template list of category(3). Here it deals with synonymous since halo and hello has the same output.

3- An atomic pattern which is in category(4) will match sentence 2, and a random answer will be selected from the template list of its category.

4- AIML interpreter will combine these answers together and display it.

Not that: <sr/> = <srai> <star/> </srai>, call recursive matcher and apply it to the input in <star/>, that is in wildcards.

1.3 Preparation for Pattern Matching

Before starting pattern matching procedures each input to the AIML interpreter must pass through two processes:
   1- Normalization Process.
   2- Producing input path from each sentence.

1.3.1 Normalization Process, involves 3 steps:

   a.   **Substitution Normalizations:**
   are heuristics applied to an input that attempts to retain information in the input that would otherwise be lost   during the sentence splitting or pattern fitting normalization. It can distinguish the dot notation if it is used as an abbreviation, end of sentences or just prefix of extension name and replace it by its appropriate meaning.

   b.   **Sentence Splitting Normalization:**
    Split input into sentences using rules like " break sentences at periods", after ?
    and !.
   Note that: Splitting is done before matching, between partitioning occurred
                during pattern matching.

   c.   **Pattern Fitting Normalization:**
    It involves two tasks:
   - Removing punctuation from input to make it compatible with speech conversation.
   - Converting input letters to upper case, this is necessary, because if origin user input was either lower or upper case it will be matched with patterns stored in capital letters but the opposite is not true.

   Note that: The normalization process must applied in the previous order and
                at least Pattern Fitting Normalization must be done.

Example:

Input	substitution normalized form	sentence-splitting normalized form	pattern-fitting normalized form
`"What time is it?"`	`"What time is it?"`	`"What time is it"`	`"WHAT TIME IS IT"`
`"Quickly, go to http://alicebot.org!"`	`"Quickly, go to http://alicebot dot org!"`	`"Quickly, go to http://alicebot dot org"`	`"QUICKLY GO TO HTTP ALICEBOT DOT ORG"`
`":-) That's funny."`	`"That is funny."`	`"That is funny"`	`"THAT IS FUNNY"`
`"I don't know. Do you, or will you, have a robots.txt file?"`	`"I do not know. Do you, or will you, have a robots dot txt file?"`	`"I do not know"` `"Do you, or will you, have a robots dot txt file"`	`"I DO NOT KNOW"` `"DO YOU OR WILL YOU HAVE A ROBOTS DOT TXT FILE"`

1.3.2 Producing Input Path for each sentence:

This path has the following form:

Input <That>Tvalue<Topic>Pvalue

It has three elements:

    a- Normalized Input.

    b- <That> tag: previous bot answer, normalized in the same way as input.
        Tvalue = previous bot answer if exists.
        Else
        Tvalue = *

    c- <Topic> tag: is an optional top level element that contains categories and has a name attribute assign to it, in order to allow interpreter to prefer responses dealing with that topic.
        Pvalue = Topic name if exists.
        Else
        Pvalue = *

Example:

Normalized input	Previous bot output (normalized)	Value of `topic` predicate	Input path
`"YES"`	`"DO YOU LIKE CHEESE"`	`""`	`"YES <that> DO YOU LIKE CHEESE <topic> *"`
`"MY NAME IS NOEL"`	`"I GUESS SO"`	`"MUSHROOMS"`	`"MY NAME IS NOEL <that> I GUESS SO <topic> MUSHROOMS"`

1.4 Pattern Matching Behaviour

AIML interpreter try to match word by word to obtain the largest pattern matching which is the best one.
This behaviour can be described in terms of the class Graphmaster which has a set of nodes called Nodemappers_ that map branches from each node_ and branches represents the first words of all patterns and for wildcards. So it is parent child relationship.

Assume the user input start with word X and the root of this tree structure is a folder of file system that contains all patterns and templates, the pattern matching algorithm applied here using depth first search techniques.

1. If the folder has a subfolder start with underscore then turn to ,"_/" , scan through it to match all words suffixed X, if no match then:

2. Go back to folder, try to find a subfolder start with word X, if so turn to "X/", scan for matching the tail of X. Patterns are matched.
   If no match then:

3. Go back to the folder, try to find a subfolder start with star notation, if so, turn to "*/", try all remaining suffixes of input following "X" to see if one match. If no match was found, change directory back to the parent of this folder, and put "X" back on the head of the input.

4. When match is found, the process stops, and the template that belongs to that category is processed by interpreter to construct the output. See graph(1).

Example

Assume the following categories:

```
(1)   <category>
      <pattern>_ WHAT IS 2 AND 2</pattern>
      <template>
      <sr/>  <srai>WHAT IS 2 AND 2</srai>
      </template>
      </category>
      <category>

(2)   <pattern>WHAT IS 2 *</pattern>
      <template>
      <random>
      <li>Two.</li>
      <li>Four.</li>
      <li>Six.</li>
      <li>12.</li>
      </random>
      </template>
      </category>
```

```
(3)    <category>
       <pattern>HALO</pattern>
       <template>
       <srai>HELLO</srai>
       </template>
       </category>

(4)    <category>
       <pattern>HELLO</pattern>
       <template>
       <random>
       <li>Well hello there!</li>
       <li>Hi there!</li>
       <li>Hi there. I was just wanting to talk</li>
       <li>Hello there !</li>
       </random>
       </template>
       </category>
```

User Input: **halo what is 2 and 2 ?**
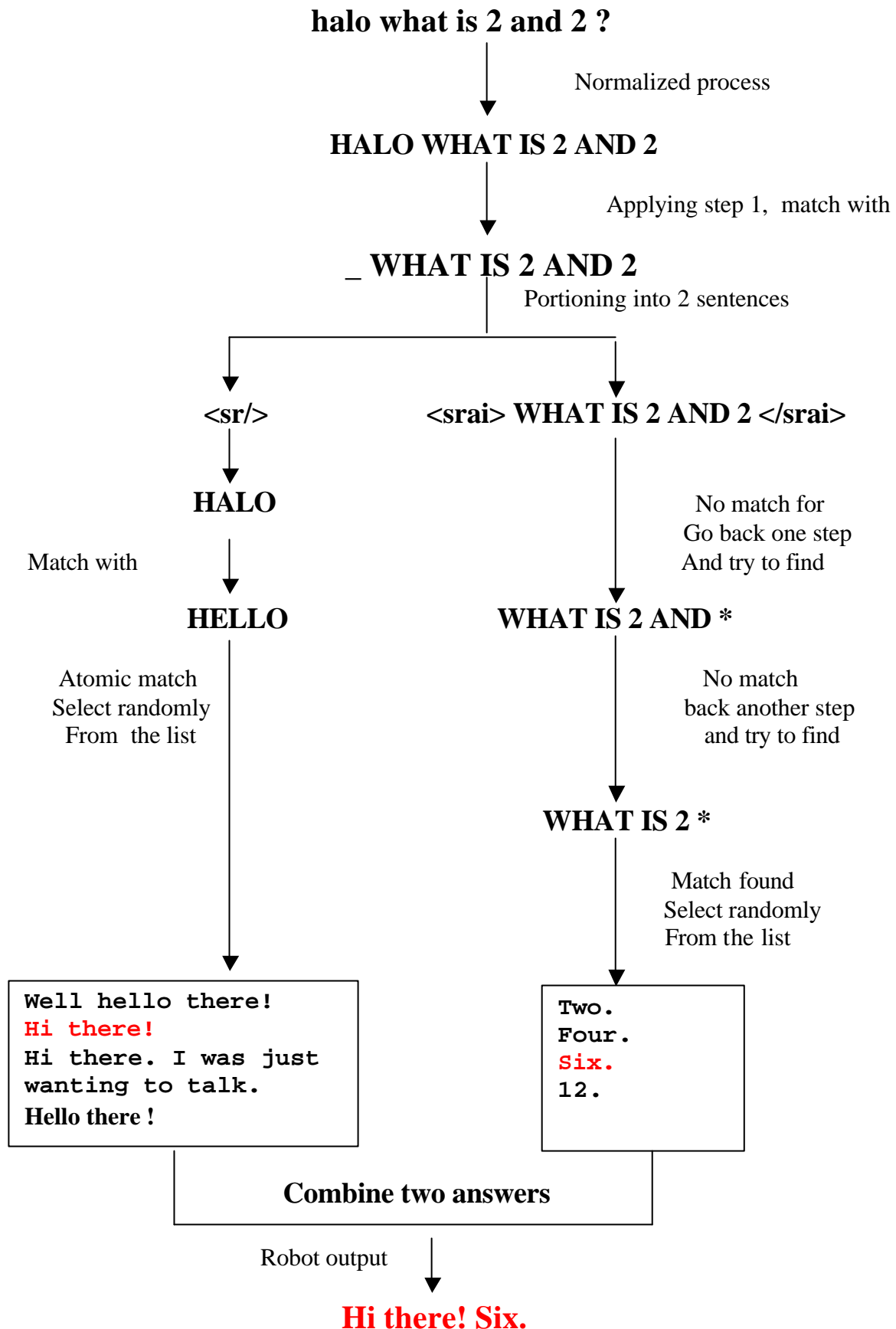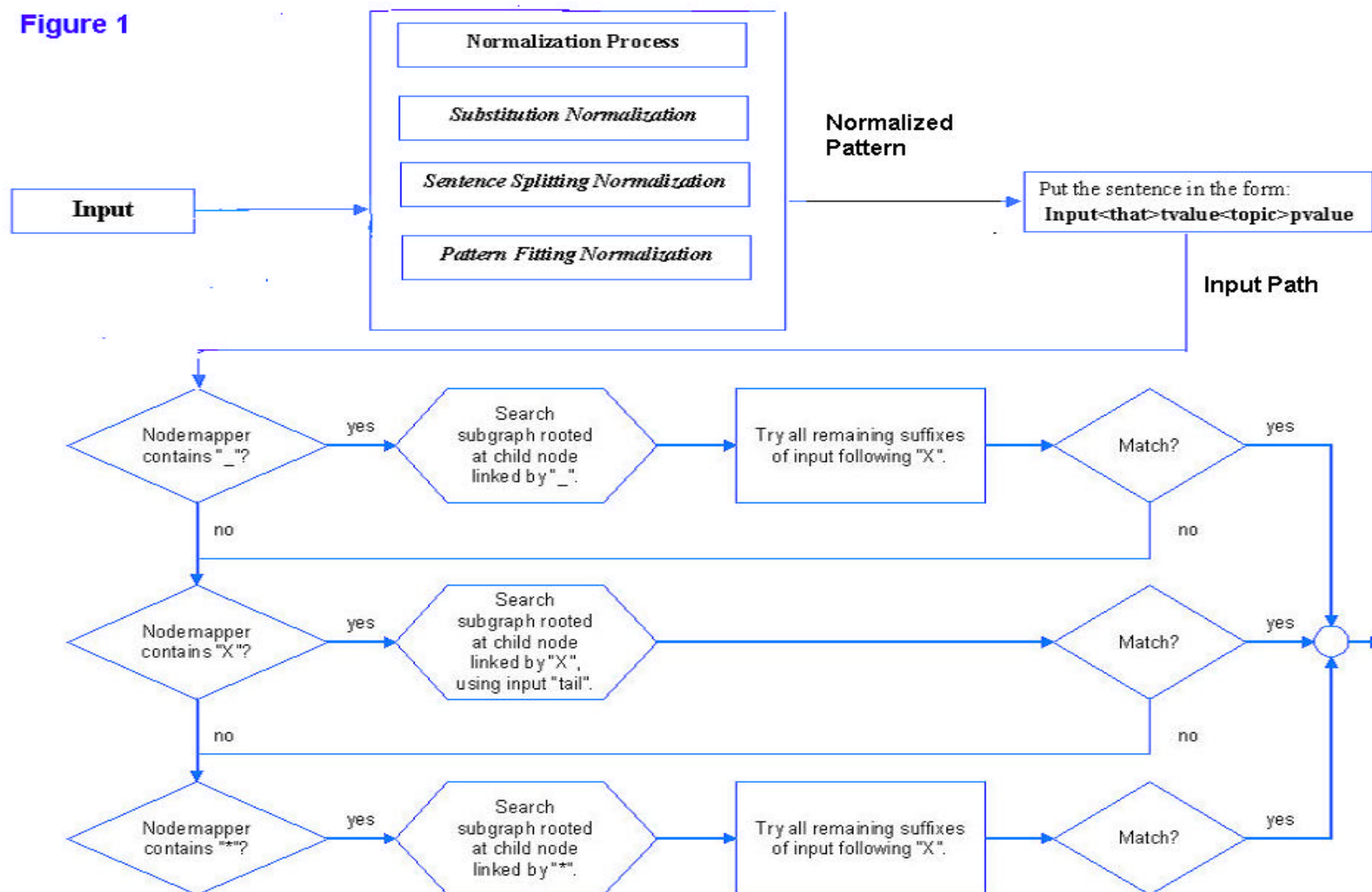*Robot output:* **Hi there! Six.**

The tree structure of the pattern matching process is:

halo what is 2 and 2 ?

↓ Normalized process

HALO WHAT IS 2 AND 2

↓ Applying step 1, match with

_ WHAT IS 2 AND 2

Portioning into 2 sentences

↓

HALO

Match with ↓

HELLO

Atomic match
Select randomly
From the list

↓

```
Well hello there!
Hi there!
Hi there. I was just
wanting to talk.
Hello there !
```

<srai> WHAT IS 2 AND 2 </srai>

No match for
Go back one step
And try to find

↓

WHAT IS 2 AND *

No match
back another step
and try to find

↓

WHAT IS 2 *

Match found
Select randomly
From the list

↓

```
Two.
Four.
Six.
12.
```

Combine two answers

Robot output ↓

Hi there! Six.

Figure 1

Input

Normalization Process

Substitution Normalization

Sentence Splitting Normalization

Pattern Fitting Normalization

Normalized Pattern

Put the sentence in the form:
Input<that>tvalue<topic>pvalue

Input Path

Nodemapper contains "_"? → yes → Search subgraph rooted at child node linked by "_". → Try all remaining suffixes of input following "X". → Match? → yes
no → Match? no

Nodemapper contains "X"? → yes → Search subgraph rooted at child node linked by "X", using input "tail". → Match? → yes
no → Match? no

Nodemapper contains "*"? → yes → Search subgraph rooted at child node linked by "*". → Try all remaining suffixes of input following "X". → Match? → yes

10

2. Elizabeth

It is an adaptation of Joseph Weizerbaum's Eliza program, in which the various selection, substitution, and phrase storage mechanisms have been enhanced and generalized to increase both flexibility and its potential adaptability.

2.1 Script File format

Knowledge is stored as a script in a text file, where each line in this text is started with a script command notation to distinguish between them, these notations are: W, Q, V, I, K, N, O, M, '& ', AND '/ ', that denotes in order, welcome message , quitting message, void input, input transformation, key word pattern, key word response pattern, output transformation, memorised phrase, action to be performed within a message, and a comment.

The script file may contain at most 4 parts as bellow:

Part One: Script command lines holding robot responses dealing with the cases of welcome, void and no key word messages.
- **Welcome messages:** begins with 'W' letter and one of them will be selected randomly by the system when it is started.
- **Void messages:** begins with 'V' letter and one of them will be selected when user input is empty, the case when the user just press enter.
- **No key word messages:** begins with 'N' letter and one of them will be selected randomly by the system when there is no keyword pattern match occur.

Example:

W HELLO, I AM Elizabeth. WHAT DO YOU LIKE TO TALK ABOUT ?
V CAN'T YOU THINK OF ANYTHING T O SAY ?
N TELL ME WHAT YOU LIKE DOING?

Part Two: Input transformation rules, that transforming user input to another form to be compatible with the defined keyword patterns set. It begins with 'I' letter denoting input.
Example:
> **I** mum => mother
> **I** dad => father

This means when there is word mum within user text, it will be replaced by the word mother, and dad will be replaced by father.

Note that: these rules must be written in lower case.

Part Three: Output transformation rules that change personal pronouns to be appropriate as a response. It begins with 'O' denoting output.

Example:

    O   i am => YOU ARE

    O   i => YOU

    O   my => YOUR

Note that: the right hand side will used in the final response so it must be in upper case, but left hand side may be lower when changing input pronoun or may be upper when dealing with robot response. These will be obvious later on within the pattern matching process.

Part Four: Key word patterns that will be used with its responses in matching process. Each keyword pattern line start with 'K' letter followed by its response that begins with 'R' letter.

There are two types of key word patterns:

- **Simple Patterns:** matching only single word

    K   MOTHER

    R   TELL ME MORE ABOUT YOUR FAMILY

    if the word 'mother' appear in the input match occur and

    the message "TELL ME MORE ABOUT YOUR FAMILY"

     will appear as a response .

- **Composite patterns:** the keyword pattern may be a sentence of phrase, word, string letter, anything and nothing.

    K   I THINK [phrase]

    R   WHY DO YOU THINK [phrase] ?

    if the user input was: I think I am ill.

    Robot output will be: WHY DO YOU THINK YOU ARE ILL ?

    [phrase] will be matched with " i am ill ". Note that the pronoun I is changed to you according to the previous output transformation rule.

Different keywords can have the same response and at the same time any keyword may have a set of responses to be selected randomly by the system.
Example:

    K   MOTHER

    K   FATHER

      R   TELL ME MORE ABOUT YOUR FAMILY?

      R   TELL ME ABOUT YOUR CHILDHOOD?

      R   ARE YOU THE YOUNGEST IN YOUR FAMILY?

So if the user enter a sentence that contains 'mother' or 'father' they will have the same response.

Note that:

- Within the Input/Output Transformation and Keyword patterns, the text pattern may contain: a letter, a string within word, a word not included punctuation, a phrase of contiguous words, anything of contiguous words and punctuations, a bracket match only anything between brackets only, any punctuation mark, and also nothing. These patterns *in previous order* are denoted by : [letter], [string], [word], [phrase], [any], [bracket],[*], and [ ]. Any text pattern must contain at least one element according to its category, but you can add '?' at the end of each one represent nothing(i.e. [word?]: one word or nothing), also these can take numbers [phras1], [phrase2], and so on.

- When the script command is stored in the system, an index code is attached to it to identify the command later on, by default the index code has 3 digit form(i.e.: '001'), and it is possible also for the user to create his own index code that simplify further actions for him.

2.2 Pattern Matching

Before starting the matching process, an active text is generated by:

1. Converting user input to lower case.
2. Inserted spaces between words and punctuation.
3. Removing some characters from the input, except:
   ! " ' ( ) , - . 0…9 : ; ? A…Z a…z

The matching process involves five phases:

1. Matching with Input Transformation Rules.
2. Matching with Keyword patterns.
3. Matching with Output transformation rules.
4. Matching with Void or No keyword messages.
5. Performing any Dynamic processes.

Phase One: Input Transformation Matching

Apply input transformation rules on the active text using the following algorithm:

Step 0: For each provided rule, in turn according to their index code:

 Step 1: Search the active text to see if it matches the left hand side of
         the rule, **IF Match** then,

   a. Replace it by the right hand side of that rule,
   b. Perform any related action or record it to be performed later on in   phase five.
   c. Repeat step 1 for the rest of the active text.
      If the same rule is applicable to the active text more than 10 times in succession, then it is applied just once.

Step 2: Go back to step 0, consider the second rule provided and
         repeat the same process.

Note that: you have to check all the input rules either match occur or not.

Phase Two: Keyword pattern matching.

Try to find a key word within the active text, using the following:

Step 0: For each keyword pattern defined, in turn, starting from the
        lower index code do:

Step 1: Try to find this keyword within the active text,
        IF Match occurs then,
         a. One response will be selected randomly from the response list related
            to this keyword pattern
         b. Record the index code of this response so it will not be used if same
            match occurs on the next occasion.
         c. Perform any related action or record it to be performed later on in
            phase five.
         d. Go to Phase Three.

   ELSE
   Go back to step 0, pick up the second keyword and repeat the process.

IF No match occurs after searching all defined keyword patterns then go to Phase
Four

Phase Three: Output transformation matching

Includes two steps:
   1- Apply the output transforming rules on the active text using the same
      algorithm in Phase One.
   2- Change the active text to upper case and display it as the robot response.

Phase Four: Since no match occur with the keyword patterns then either the active
text is:
         a. Void _empty_, and one of the void messages denoted by 'V' will be
            selected as a robot response.

            OR

         b. No keyword found within the active text, in this case a response will be
            selected randomly from No keyword messages denoted by 'N'.

Phase Five: Perform any Recorded Dynamic Process
perform the assigned actions in phase one and three if exists. This process will be
discussed   in the next section.

Note that:
   • If two keyword patterns exists in the same active text, then the response is
     selected according to keyword that has less index code.
   • If the same input reoccurs within a single conversation then if possible the
     system will attempt to make a different random choice from one previously
     made.

Example

Assume you have the following script file:

/ The Script begins with Welcome, Void, No-Keyword and Quit responses:

'001' **W** HELLO, I'M Elizabeth. WHAT WOULD YOU LIKE TO TALK
          ABOUT?
'001' **V** CAN'T YOU THINK OF ANYTHING TO SAY?
'002' **V** ARE YOU ALWAYS CHIE ?
'001' **N** TELL ME WHAT YOU LIKE DOING.
'001' **Q** GOODBYE! DO COME BACK SOON.

/ Next come the Input transformations:

'001' **I** mum => mother
'002' **I** dad => father

/ Then the Output transformations:

'001' **O** i am => YOU ARE
'002' **O** you are => I AM
'003' **O** my => YOUR
'004' **O** your => MY
'005' **O** me => YOU
'006' **O** I IS => I AM
'007' **O** YOU IS => YOU ARE

/ And four groups of Keyword transformations:

'001' **K** I THINK [phrase]
          '001' **R** WHY DO YOU THINK [phrase] ?
'002' **K** MOTHER
'003' **K** FATHER
          '001'   **R** TELL ME MORE ABOUT YOUR FAMILY.
          '002'   **R** WHAT DO YOU REMEMBER MOST ABOUT YOUR
                   CHILDHOOD?
          '003'   **R** ARE YOU THE YOUNGEST IN YOUR FAMILY?

'004' **K** [phrase1] IS YOUNGER THAN [phrase2]
          '001' **R**   SO [phrase2] IS OLDER THAN [phrase1] .

'005' **K** I LIKE [string]ING
          '001'   **R** HAVE YOU [string]ED AT ALL RECENTLY?

'006' **K**   [ ] my [phrase]
        & {M [phrase]}
        '001' **R**   YOUR [phrase] ?

15

The following trace using the previous script to illustrate different inputs with its responds:

User Input	I nput Transforming	Keyword Patterns	Output Transforming	Respond	Actions
Dad loves Mum	Match: '001' mum => mother '002' dad => father	Match: '001'K MOTHER '001' R TELL ME MORE ABOUT YOUR FAMILY.	-	TELL ME MORE ABOUT YOUR FAMILY.	-
My sister is a teacher.	-	Match: '006' K my [phrase] [phrase] => sister is a teacher '001' R YOUR is a teacher?	-	YOUR SISTER IS A TEACHER?	'fam' M sister is a teacher
My brother is younger than me.	-	Match: '004' K [phrs1] IS YOUNGER THAN [phrs2] [phrs1] => my brother [phrs2] => me '001' R SO [phrs2] IS OLDER THAN [phrs1].	Match: [phrs1] with '003' my => YOUR [phrs2] with '005' me => YOU	SO YOU IS OLDER THAN YOUR BROTHER.	-
I like reading	-	Match '005' K I LIKE [string]ING [string] => read '001' R HAVE YOU [string]ED AT ALL RECENTLY?	-	HAVE YOU READED AT ALL RECENTLY?	-

2.3 Dynamic Process

Performing a set of actions that modify the script while the conversation is in progress. These actions are included within curly brackets that is preceding by '&' symbol.

Its format is: & { Script Command } and you can have as many deep as you want.

These actions includes adding, memorization and deleting script commands.

1. Adding new Script command:

This action will add a new script command if it is not exist and replace it by new one if it is already found.

Example:

I my sister => my sister
   & { K MOTHER
          & { N DOES ANYTHING ELSE ABOUT YOUR MOTHER ? }
       R HOW WELL YOUR MOTHER AND SISTER GET ON? }

There are two actions to be performed in the following sequence:

Action 1:

When matching occur with the input rule, my sister, it will remain the same and giving answer as appropriate, at the same time the system will search for the keyword " MOTHER", if find then Add the response " How Well YOUR Mother AND SISTER GET ON?" to its response list, and record the action inside the inner curly brackets.
ELSE
add the keyword "mother" and its response to the end of the keyword patterns group and record the inner action.

Action 2:

When matching occur with "MOTHER", then a no keyword message will add it to the list or replace the old one.

2. Memorization a script command

This is used to hold the user input or robot response for further using during the conversation.
This script command is denoted by 'M' letter.

Example:

K [ ] MY [PHRASE]
    & { M fam [PHRASE] ?
 R YOUR [PHRASE] ?

[ ] means that "my" must be at the beginning of the sentence.

IF the user enter : my brother is beautiful.
Robot answer: YOUR BROTHER IS BEATIFUL?
Action performed: memorizing [brother is beautiful] and associate it
with "fam" as his index code.

3. Deleting Script Command:

Delete and script command by using back slash notation.

**V **   : Deletes all current void input messages.
**M ** fam   : deletes the memorization command with index "fam".

Note that
- This process is generated according to a specific order, they are performed only after the user's input has been fully processed to generate an appropriate output.
- The memorized actions always precede any other action.

2.4 Implementing Grammatical Rules

Elizabeth has the ability to produce a grammar structure analysis of a sentence using set of input transformation rules to represent certain grammar. This provide an introduction to some of the major concepts and techniques of natural language processing.

Consider the following grammar:

S => NP VP
NP => D N
VP => V NP

It can be represented by the input rules to deal with certain nouns, verbs and determiners, as follows:

I a => (A d)
I the => (THE d)
I cat => (CAT n)
I dog => (DOG n)
I rabbit => (RABBIT n)
I bites => (BITES v)
I chases => (CHASES v)
I likes => (LIKES v)
I ([brak1] d) ([brak2] n) => (([brak1] D) ([brak2] N) np)
I ([brak1] v) ([brak2] np) => (([brak1] V) ([brak2] NP) vp)

I ([brak1] np) ([brak2] vp) => (([brak1] NP) ([brak2] VP) s)
K [any?]
R [any?]

W TYPE A SENTENCE USING: A, THE, CAT, DOG, RABBIT, BITES,
    CHASES, LIKES

This script starts from the Welcome message, so

Robot: TYPE A SENTENCE USING: A, THE, CAT, DOG, RABBIT, BITES,
        CHASES, LIKES
user input is: the cat likes the dog
Response : (((THE D) (CAT N) NP) ((LIKES V) ((THE D) (DOG N) NP) VP) s)

This answer is given according to the following trace:

(THE d) (CAT n) (LIKES v) (THE d) (DOG n)
((THE D) (CAT N) np) (LIKES v) ((THE D) (DOG N) np)
((THE D) (CAT N) np) ((LIKES V) ((THE D) (DOG N) NP) vp)
(((THE D) (CAT N) NP) ((LIKES V) ((THE D) (DOG N) NP) VP) s)

3. A Comparison between ALICE and Elizabeth

From the previous two sections, you can determine the following:

- ALICE used a simple pattern template to represent input and output,
  and also using simple pattern matching algorithm.
  Between Elizabeth uses Input rules, keyword patterns and output
  rules to generate a response.

- The recursive techniques used in ALICE is considered as a power
  point of the system, it is used for simplifying the input by calling
  match categories recursively. Contradictory, the nature of some rules
  in Elizabeth may cause cycling or iteration, which is solved by
  applying the rule only once if it is applicable for the active text more
  than 10, times in succession.

- In ALICE there is the ability to combine two answers in the case of
  splitting happened within Normalization Process, or the partitioning
  caused by the recursive process, The recursive process provide a
  way to partition the sentence to two sentences then combine their
  results which is not available by Elizabeth.

- The most important and strong issue in ALICE is the pattern
  matching algorithm, which is easy and depend on depth first search.
  This algorithm try to find the longest pattern matching between

Elizabeth gives the response according to the first Keyword pattern matched.

- Both systems can change personal pronouns, a lot of complicated appear in Elizabeth related to writing some rules in upper case and others in lower, which may cause a lot of errors and give unsuitable answers. Also both systems allowed memorization for the previous input, output for further using, but Elizabeth allow other actions to occur while the conversation is under progress, that is called the dynamic process like adding, modifying and deleting script commands.

- If the same input repeat during the conversation, Elizabeth try to give different answers by using different random selection responses from the respond list. In ALICE sometimes give you the same answers or different one.

- Elizabeth has the ability to give the derivation structure for a sentence using the grammatical analysis which is not provided by ALICE.

- Elizabeth allow the user to create his own script files and it incorporates analysis tables for all steps in matching, helping the user to understand how this answer is generated.

4. Conclusion

Both systems are chatbots systems that are adopted form Eliza program, and each one has advantages and disadvantages. Alice stored a huge corpus text and Elizabeth provides the grammatical analysis for sentences, both of them give a good introduction To Artificial Intelligence, Natural Language Understanding and Patten Matching. The main inference point is it will be easier to build a machine learning for ALICE since it uses simple patterns, templates to represent input and output. Elizabeth uses more complex rules for which you need to write Input transformation, Output transformation and keyword patterns to represent a user input and Elizabeth answer. You can do that by simple pattern template using ALICE. Another important point used by ALICE is the ability to partition the user input or splitting it into two sentences and then combines the answers. This is an important issue in language processing. Elizabeth does not provide this facility. According to Elizabeth structure it will be difficult to do splitting.

5. References

1. ' Alice and Aiml: a powerful Alternative to HTML and voice XML' XML Journal, vol. 2, issue 10, www.XML-Journal.com

2. Cara Feinberg, ' Frontiers of Free Marketing', The American Prospect vol. 12 no. 14, August 13, 2001.

3. David Pescovitz, ' Look What's Talking: Software Robots', The New York Times, March 18, 1999.

4. Justin Hunt, ' The King is ready for a chat', Guardian, Thursday February 1, 2001.

5. Noel Bush, (2001) 'Artificial Intelligence Mark up Language (AIML) version 1.0.1'
   http://alice.sunlitsurf.com/TR/2001/WD-aiml/

6. Peter Millican, (2002) Elizabeth Help Files.
7. Richard S. Wallace, (200) 'Symbolic Reduction in AIML'
   http://alice.sunlitsurf.com/documentation/srai.html

8. Richard S. Wallace, (2001) 'AIML Pattern Matching Simplified'
   http://alice.sunlitsurf.com/documentation/matching.html

9. Richard S. Wallace, 'How It All Started'
   http://alice.sunlitsurf.com/articles/wallace/start.html

10. Richard S. Wallace, 'From Eliza to ALICE'
    http://alice.sunlitsurf.com/articles/wallace/eliza.html

11. Thomas Ringate, (2001) 'AIML Primer'
    http://alice.sunlitsurf.com/documentation/aiml-primer.html

12. Thomas Ringate, (2001) 'AIML Reference Manual'
    http://alice.sunlitsurf.com/documentation/aiml-reference.html